Safe Non-blocking Synchronization in Ada 202x

Johann Blieberger and Bernd Burgstaller

Institute of Computer Engineering, Automation Systems Group, TU Wien, Austria

Department of Computer Science, Yonsei University, Korea

19.6.2018

Safe Non-blocking Synchronization in Ada 202x

Johann Blieberger and Bernd Burgstaller

Institute of Computer Engineering, Automation Systems Group, TU Wien, Austria

Department of Computer Science, Yonsei University, Korea

19.6.2018

Synchronization

- mutual exclusion locks

Synchronization

- mutual exclusion locks
- Ada's protected objects (POs)

Synchronization

- mutual exclusion locks
- Ada's protected objects (POs)
- Entries and procedures of a PO execute one after another

Synchronization

- mutual exclusion locks
- Ada's protected objects (POs)
- Entries and procedures of a PO execute one after another
- makes it straight-forward for programmers to reason about updates to the shared data encapsulated by a PO

Sequential Consistency

- mutual-exclusion property of (highly-contended) **locks** stands in the way to **scalability of parallel programs** on many-core architectures

Sequential Consistency

- mutual-exclusion property of (highly-contended) **locks** stands in the way to **scalability of parallel programs** on many-core architectures
- locks do not allow **progress guarantees**, because a task may fail inside a critical section, e.g., by entering an endless loop, and thereby prevent other tasks from accessing shared data

Sequential Consistency

- mutual-exclusion property of (highly-contended) **locks** stands in the way to **scalability of parallel programs** on many-core architectures
- locks do not allow **progress guarantees**, because a task may fail inside a critical section, e.g., by entering an endless loop, and thereby prevent other tasks from accessing shared data
- $\Rightarrow$ allow method calls to **overlap** in time

Sequential Consistency

- mutual-exclusion property of (highly-contended) **locks** stands in the way to **scalability of parallel programs** on many-core architectures
- locks do not allow **progress guarantees**, because a task may fail inside a critical section, e.g., by entering an endless loop, and thereby prevent other tasks from accessing shared data
- $\Rightarrow$ allow method calls to **overlap** in time
- $\Rightarrow$ synchronization on a **finer granularity** within a method's code, via atomic *read-modify-write* (RMW) operations

Sequential Consistency

- mutual-exclusion property of (highly-contended) **locks** stands in the way to **scalability of parallel programs** on many-core architectures
- locks do not allow **progress guarantees**, because a task may fail inside a critical section, e.g., by entering an endless loop, and thereby prevent other tasks from accessing shared data
- $\Rightarrow$ allow method calls to **overlap** in time
- $\Rightarrow$ synchronization on a **finer granularity** within a method's code, via atomic *read-modify-write* (RMW) operations
- atomic operations are provided either by the CPU's instruction set architecture (ISA), or the language run-time (with the help of the CPU's ISA)

Sequential Consistency

- mutual-exclusion property of (highly-contended) **locks** stands in the way to **scalability of parallel programs** on many-core architectures
- locks do not allow **progress guarantees**, because a task may fail inside a critical section, e.g., by entering an endless loop, and thereby prevent other tasks from accessing shared data
- $\Rightarrow$ allow method calls to **overlap** in time
- $\Rightarrow$ synchronization on a **finer granularity** within a method's code, via atomic *read-modify-write* (RMW) operations
- atomic operations are provided either by the CPU's instruction set architecture (ISA), or the language run-time (with the help of the CPU's ISA)
- e.g., CAS compare&swap operation

Sequential Consistency

- mutual-exclusion property of (highly-contended) **locks** stands in the way to **scalability of parallel programs** on many-core architectures
- locks do not allow **progress guarantees**, because a task may fail inside a critical section, e.g., by entering an endless loop, and thereby prevent other tasks from accessing shared data
- $\Rightarrow$ allow method calls to **overlap** in time
- $\Rightarrow$ synchronization on a **finer granularity** within a method's code, via atomic *read-modify-write* (RMW) operations
- atomic operations are provided either by the CPU's instruction set architecture (ISA), or the language run-time (with the help of the CPU's ISA)
- e.g., CAS compare&swap operation
- **sequential consistency** ensures that method calls act as if they occurred in a **sequential, total order** that is **consistent with the program order** of each participating task

Non-blocking Synchronization Techniques

- difficult to implement

Non-blocking Synchronization Techniques

- difficult to implement
- the design of non-blocking data structures is an area of active research

Non-blocking Synchronization Techniques

- difficult to implement
- the design of non-blocking data structures is an area of active research
- a programming language must provide a strict memory model

Lock-free Synchronization – Example

```
-- Initial values:
Flag := False;
Data := 0;
```

```
1 -- Task 1:
2 Data := 1;
3 Flag := True;
```

Lock-free Synchronization – Example

```
-- Initial values:
Flag := False;
Data := 0;
```

```
1 -- Task 1:
2 Data := 1;
3 Flag := True;
```

```
1 -- Task 2:
2 loop
3   R1 := Flag;
4   exit when R1;
5 end loop;
6 R2 := Data;
```

Lock-free Synchronization – Example

```
-- Initial values:
Flag := False;
Data := 0;
```

```
1 -- Task 1:
2 Data := 1;
3 Flag := True;
```

```
1 -- Task 2:
2 loop
3   R1 := Flag;
4   exit when R1;
5 end loop;
6 R2 := Data;
```

store–store re-ordering of the assignments in lines 2 and 3 of Task 1 $\Rightarrow$ reading `R2 = 0` in Line 6 of Task 2.

Lock-free Synchronization – Example

```
-- Initial values:
Flag := False;
Data := 0;
```

```
1 -- Task 1:
2 Data := 1;
3 Flag := True;
```

```
1 -- Task 2:
2 loop
3   R1 := Flag;
4   exit when R1;
5 end loop;
6 R2 := Data;
```

store–store re-ordering of the assignments in lines 2 and 3 of Task 1 $\Rightarrow$ reading R2 = 0 in Line 6 of Task 2.

```
1 Data : Integer with Volatile; -- Ada2012
2 Flag : Boolean with Atomic; -- Ada2012
```

Ada's Volatile Variables

- guarantee that all tasks agree on the same order of updates

Ada's Volatile Variables

- guarantee that all tasks agree on the same order of updates
- $\Rightarrow$ sequentially consistent

Ada's Volatile Variables

- guarantee that all tasks agree on the same order of updates
- $\Rightarrow$ sequentially consistent
- however: relaxed SC for the sake of performance in contemporary CPU architectures

Memory (Consistency) Model

- ideally, all read/write operations of a program's tasks are SC

Memory (Consistency) Model

- ideally, all read/write operations of a program's tasks are SC
- however, the hardware memory models provided by contemporary CPU architectures **relax SC** for the sake of performance

Memory (Consistency) Model

- ideally, all read/write operations of a program's tasks are SC
- however, the hardware memory models provided by contemporary CPU architectures **relax SC** for the sake of performance
- enforcing SC on such architectures may incur a noticeable **performance penalty**

Memory (Consistency) Model

- ideally, all read/write operations of a program's tasks are SC
- however, the hardware memory models provided by contemporary CPU architectures **relax SC** for the sake of performance
- enforcing SC on such architectures may incur a noticeable **performance penalty**
- workable middle-ground between **intuition** (SC) and **performance** (relaxed hardware memory models) has been established with SC for data race-free programs (SC-for-DRF)

Memory (Consistency) Model

- ideally, all read/write operations of a program's tasks are SC
- however, the hardware memory models provided by contemporary CPU architectures **relax SC** for the sake of performance
- enforcing SC on such architectures may incur a noticeable **performance penalty**
- workable middle-ground between **intuition** (SC) and **performance** (relaxed hardware memory models) has been established with SC for data race-free programs (SC-for-DRF)
- "SC-for-DRF" requires programmers to ensure that programs are free of data races under SC

Memory (Consistency) Model

- ideally, all read/write operations of a program's tasks are SC
- however, the hardware memory models provided by contemporary CPU architectures **relax SC** for the sake of performance
- enforcing SC on such architectures may incur a noticeable **performance penalty**
- workable middle-ground between **intuition** (SC) and **performance** (relaxed hardware memory models) has been established with SC for data race-free programs (SC-for-DRF)
- "SC-for-DRF" requires programmers to ensure that programs are free of data races under SC
- $\Rightarrow$ the relaxed memory model of a SC-for-DRF CPU guarantees SC for all executions of such a program

SC-for-DRF

- on the programming language level to guarantee DRF, means for synchronization (ordering operations) have to be provided

SC-for-DRF

- on the programming language level to guarantee DRF, means for synchronization (ordering operations) have to be provided
- Ada's POs are well-suited for this purpose

SC-for-DRF

- on the programming language level to guarantee DRF, means for synchronization (ordering operations) have to be provided
- Ada's POs are well-suited for this purpose
- for non-blocking synchronization, atomic operations can be used to enforce an ordering between the memory accesses of two tasks

SC-for-DRF

- on the programming language level to guarantee DRF, means for synchronization (ordering operations) have to be provided
- Ada's POs are well-suited for this purpose
- for non-blocking synchronization, atomic operations can be used to enforce an ordering between the memory accesses of two tasks
- add language features to Ada such that atomic operations can be employed with DRF programs

Hardware support for Lock-free Synchronization

- ISAs provide atomic load/store instructions only for a limited set of primitive types

Hardware support for Lock-free Synchronization

- ISAs provide atomic load/store instructions only for a limited set of primitive types
- memory fences provide means for ordering memory operations

Hardware support for Lock-free Synchronization

- ISAs provide atomic load/store instructions only for a limited set of primitive types
- memory fences provide means for ordering memory operations
- a memory fence requires that all memory operations before the fence (in program order) must be committed to the memory hierarchy before any operation after the fence

Hardware support for Lock-free Synchronization

- ISAs provide atomic load/store instructions only for a limited set of primitive types
- memory fences provide means for ordering memory operations
- a memory fence requires that all memory operations before the fence (in program order) must be committed to the memory hierarchy before any operation after the fence
- then, for data to be transferred from one thread to another it is not necessary to be atomic anymore

Lock-free Synchronization – Example revisited

```
-- Initial values:
Flag := False;
Data := 0;
```

```
1 -- Task 1:
2 Data := 1;
3 -- memory fence;
4 Flag := True;
```

Lock-free Synchronization – Example revisited

```
-- Initial values:
Flag := False;
Data := 0;
```

```
1 -- Task 1:
2 Data := 1;
3 -- memory fence;
4 Flag := True;
```

```
1 -- Task 2:
2 loop
3   R1 := Flag;
4   exit when R1;
5 end loop;
6 -- memory fence;
7 R2 := Data;
```

Lock-free Synchronization – Example revisited

```
-- Initial values:
Flag := False;
Data := 0;
```

```
1 -- Task 1:
2 Data := 1;
3 -- memory fence;
4 Flag := True;
```

```
1 -- Task 2:
2 loop
3   R1 := Flag;
4   exit when R1;
5 end loop;
6 -- memory fence;
7 R2 := Data;
```

```
1 Data : Integer;
2 Flag : Boolean with Atomic;
```

Concurrent Objects

- define *non-blocking* **concurrent objects** similar to protected objects

Concurrent Objects

- define *non-blocking* **concurrent objects** similar to protected objects
- **entries** of concurrent objects will not block on **guards**;

Concurrent Objects

- define *non-blocking* **concurrent objects** similar to protected objects
- **entries** of concurrent objects will not block on **guards**; they will **spin** until the guard evaluates to true

Concurrent Objects

- define *non-blocking* **concurrent objects** similar to protected objects
- **entries** of concurrent objects will not block on **guards**; they will **spin** until the guard evaluates to true
- **functions**, **procedures**, and **entries** of concurrent objects are allowed to execute and to modify the encapsulated data **in parallel**

Concurrent Objects

- define *non-blocking* **concurrent objects** similar to protected objects
- **entries** of concurrent objects will not block on **guards**; they will **spin** until the guard evaluates to true
- **functions**, **procedures**, and **entries** of concurrent objects are allowed to execute and to modify the encapsulated data **in parallel**
- **private entries** for concurrent objects are supported

Concurrent Objects

- define *non-blocking* **concurrent objects** similar to protected objects
- **entries** of concurrent objects will not block on **guards**; they will **spin** until the guard evaluates to true
- **functions**, **procedures**, and **entries** of concurrent objects are allowed to execute and to modify the encapsulated data **in parallel**
- **private entries** for concurrent objects are supported
- concurrent objects will use **synchronized types** for synchronizing data access

Concurrent Objects

- define *non-blocking* **concurrent objects** similar to protected objects
- **entries** of concurrent objects will not block on **guards**; they will **spin** until the guard evaluates to true
- **functions**, **procedures**, and **entries** of concurrent objects are allowed to execute and to modify the encapsulated data **in parallel**
- **private entries** for concurrent objects are supported
- concurrent objects will use **synchronized types** for synchronizing data access
- **aspect** `Synchronized_Components` (similar to Ada2012's aspect `atomic`, ...)

Memory Order and Constraints for Compilers and CPUs

- **relaxed:** no inter-thread constraints

Memory Order and Constraints for Compilers and CPUs

- **relaxed:** no inter-thread constraints
- **release/acquire:** writing thread releases the data / reading thread acquires the data

Memory Order and Constraints for Compilers and CPUs

- **relaxed:** no inter-thread constraints
- **release/acquire:** writing thread releases the data / reading thread acquires the data
- **sequentially consistent:** all threads observe the same, total order of operations

Memory Order and Constraints for Compilers and CPUs

- **relaxed:** no inter-thread constraints
- **release/acquire:** writing thread releases the data / reading thread acquires the data
- **sequentially consistent:** all threads observe the same, total order of operations

semantics are enforced by compiler and CPU, e.g. via memory fences

Synchronized Variables

- **aspect** `Synchronized` (inside of COs)

Synchronized Variables

- **aspect** `Synchronized` (inside of COs)
- **read** accesses labeled via **attribute** `Concurrent_Read`

Synchronized Variables

- **aspect** `Synchronized` (inside of COs)
- **read** accesses labeled via **attribute** `Concurrent_Read`
- **write** accesses labeled via **attribute** `Concurrent_Write`

Synchronized Variables

- **aspect** `Synchronized` (inside of COs)
- **read** accesses labeled via **attribute** `Concurrent_Read`
- **write** accesses labeled via **attribute** `Concurrent_Write`
- parameter `Memory_Order`
  - `Sequentially_Consistent` (default)

Synchronized Variables

- **aspect** `Synchronized` (inside of COs)
- **read** accesses labeled via **attribute** `Concurrent_Read`
- **write** accesses labeled via **attribute** `Concurrent_Write`
- parameter `Memory_Order`
  - `Sequentially_Consistent` (default)
  - `Acquire` (only for reads)

Synchronized Variables

- **aspect** `Synchronized` (inside of COs)
- **read** accesses labeled via **attribute** `Concurrent_Read`
- **write** accesses labeled via **attribute** `Concurrent_Write`
- parameter `Memory_Order`
  - `Sequentially_Consistent` (default)
  - `Acquire` (only for reads)
  - `Release` (only for writes)

Synchronized Variables

- **aspect** `Synchronized` (inside of COs)
- **read** accesses labeled via **attribute** `Concurrent_Read`
- **write** accesses labeled via **attribute** `Concurrent_Write`
- parameter `Memory_Order`
  - `Sequentially_Consistent` (default)
  - `Acquire` (only for reads)
  - `Release` (only for writes)
  - `Relaxed`

Synchronized Variables – Examples

- ```
  X: integer with Synchronized;
  Y: integer with Synchronized;
  ...
  X'Concurrent_Write(Memory_Order => Release) :=
                        Y'Concurrent_Read(Memory_Order => Acquire);
  ```
  ```

Synchronized Variables – Examples

- ```
  X: integer with Synchronized;
  Y: integer with Synchronized;
  ...
  X'Concurrent_Write(Memory_Order => Release) :=
                        Y'Concurrent_Read(Memory_Order => Acquire);
  ```

- **variable specific default values** via aspects

  ```
  X: integer with Synchronized, Memory_Order_Write => Release;
  Y: integer with Synchronized, Memory_Order_Read => Acquire;
  ...
  X := Y;
  ```
  ```

Read-Modify-Write Variables

- e.g. mapped to *compare&swap* operations (CAS)

Read-Modify-Write Variables

- e.g. mapped to *compare&swap* operations (CAS)
- `CAS(variable, new_value, expected_value)`

Read-Modify-Write Variables

- e.g. mapped to *compare&swap* operations (CAS)
- `CAS(variable, new_value, expected_value)`
- **aspect** `Read_Modify_Write` $\Rightarrow$ **aspect** `Synchronized`

Read-Modify-Write Variables

- e.g. mapped to *compare&swap* operations (CAS)
- `CAS(variable, new_value, expected_value)`
- **aspect** `Read_Modify_Write` $\Rightarrow$ **aspect** `Synchronized`
- **write** access via the **attribute** `Concurrent_Exchange`

Read-Modify-Write Variables

- e.g. mapped to *compare&swap* operations (CAS)
- `CAS(variable, new_value, expected_value)`
- **aspect** `Read_Modify_Write` $\Rightarrow$ **aspect** `Synchronized`
- **write** access via the **attribute** `Concurrent_Exchange`
- parameters `Memory_Order_Success` and `Memory_Order_Failure`

Read-Modify-Write Variables

- e.g. mapped to *compare&swap* operations (CAS)
- `CAS(variable, new_value, expected_value)`
- **aspect** `Read_Modify_Write` $\Rightarrow$ **aspect** `Synchronized`
- **write** access via the **attribute** `Concurrent_Exchange`
- parameters `Memory_Order_Success` and `Memory_Order_Failure`
  - `Sequentially_Consistent` (default)

Read-Modify-Write Variables

- e.g. mapped to *compare&swap* operations (CAS)
- `CAS(variable, new_value, expected_value)`
- **aspect** `Read_Modify_Write` $\Rightarrow$ **aspect** `Synchronized`
- **write** access via the **attribute** `Concurrent_Exchange`
- parameters `Memory_Order_Success` and `Memory_Order_Failure`
  - `Sequentially_Consistent` (default)
  - `Acquire`

Read-Modify-Write Variables

- e.g. mapped to *compare&swap* operations (CAS)
- `CAS(variable, new_value, expected_value)`
- **aspect** `Read_Modify_Write` $\Rightarrow$ **aspect** `Synchronized`
- **write** access via the **attribute** `Concurrent_Exchange`
- parameters `Memory_Order_Success` and `Memory_Order_Failure`
  - `Sequentially_Consistent` (default)
  - `Acquire`
  - `Release` (success only)

Read-Modify-Write Variables

- e.g. mapped to *compare&swap* operations (CAS)
- `CAS(variable, new_value, expected_value)`
- **aspect** `Read_Modify_Write` ⇒ **aspect** `Synchronized`
- **write** access via the **attribute** `Concurrent_Exchange`
- parameters `Memory_Order_Success` and `Memory_Order_Failure`
  - `Sequentially_Consistent` (default)
  - `Acquire`
  - `Release` (success only)
  - `Relaxed`

Read-Modify-Write Variables

- e.g. mapped to *compare&swap* operations (CAS)
- `CAS(variable, new_value, expected_value)`
- **aspect** `Read_Modify_Write` ⇒ **aspect** `Synchronized`
- **write** access via the **attribute** `Concurrent_Exchange`
- parameters `Memory_Order_Success` and `Memory_Order_Failure`
  - `Sequentially_Consistent` (default)
  - `Acquire`
  - `Release` (success only)
  - `Relaxed`
- **read** access via **attribute** `Concurrent_Read`

Read-Modify-Write Variables

- e.g. mapped to *compare&swap* operations (CAS)
- `CAS(variable, new_value, expected_value)`
- **aspect** `Read_Modify_Write` $\Rightarrow$ **aspect** `Synchronized`
- **write** access via the **attribute** `Concurrent_Exchange`
- parameters `Memory_Order_Success` and `Memory_Order_Failure`
  - `Sequentially_Consistent` (default)
  - `Acquire`
  - `Release` (success only)
  - `Relaxed`
- **read** access via **attribute** `Concurrent_Read`
- parameter `Memory_Order`

Read-Modify-Write Variables

- e.g. mapped to *compare&swap* operations (CAS)
- `CAS(variable, new_value, expected_value)`
- **aspect** `Read_Modify_Write` $\Rightarrow$ **aspect** `Synchronized`
- **write** access via the **attribute** `Concurrent_Exchange`
- parameters `Memory_Order_Success` and `Memory_Order_Failure`
  - `Sequentially_Consistent` (default)
  - `Acquire`
  - `Release` (success only)
  - `Relaxed`
- **read** access via **attribute** `Concurrent_Read`
- parameter `Memory_Order`
  - `Sequentially_Consistent` (default)

Read-Modify-Write Variables

- e.g. mapped to *compare&swap* operations (CAS)
- `CAS(variable, new_value, expected_value)`
- **aspect** `Read_Modify_Write` $\Rightarrow$ **aspect** `Synchronized`
- **write** access via the **attribute** `Concurrent_Exchange`
- parameters `Memory_Order_Success` and `Memory_Order_Failure`
  - `Sequentially_Consistent` (default)
  - `Acquire`
  - `Release` (success only)
  - `Relaxed`
- **read** access via **attribute** `Concurrent_Read`
- parameter `Memory_Order`
  - `Sequentially_Consistent` (default)
  - `Acquire`

Read-Modify-Write Variables

- e.g. mapped to *compare&swap* operations (CAS)
- `CAS(variable, new_value, expected_value)`
- **aspect** `Read_Modify_Write` ⇒ **aspect** `Synchronized`
- **write** access via the **attribute** `Concurrent_Exchange`
- parameters `Memory_Order_Success` and `Memory_Order_Failure`
  - `Sequentially_Consistent` (default)
  - `Acquire`
  - `Release` (success only)
  - `Relaxed`
- **read** access via **attribute** `Concurrent_Read`
- parameter `Memory_Order`
  - `Sequentially_Consistent` (default)
  - `Acquire`
  - `Relaxed`

Example Lock-free Stack (1/2)

```
subtype Data is Integer;

type List;
type List_P is access List;
type List is
  record
    D: Data;
    Next: List_P;
  end record;

Empty: exception;

concurrent Lock_Free_Stack
is
  entry Push(D: Data);
  entry Pop(D: out Data);
private
  Head: List_P with Read_Modify_Write,
    Memory_Order_Read => Relaxed,
    Memory_Order_Write_Success => Release,
    Memory_Order_Write_Failure => Relaxed;
end Lock_Free_Stack;
```

Example Lock-free Stack (2/2)

```
concurrent body Lock_Free_Stack is
  entry Push (D: Data)
      until Head = Head'OLD is
    New_Node: List_P := new List;
  begin
    New_Node.all := (D => D, Next => Head);
    Head := New_Node; -- RMW
  end Push;

  entry Pop(D: out Data)
      until Head = Head'OLD is
    Old_Head: List_P;
  begin
    Old_Head := Head;
    if Old_Head /= null then
      Head := Old_Head.Next; -- RMW
      D := Old_head.D;
    else
      raise Empty;
    end if;
  end Pop;
end Lock_Free_Stack;
```

Example – Generic Release-Acquire Object (1/2)

```
generic
   type Data is private;
package Generic_Release_Acquire is

   concurrent RA
   is
      procedure Write (d: Data);
      entry Get (D: out Data);
   private
      Ready: Boolean := false with Synchronized,
        Memory_Order_Read => Acquire,
        Memory_Order_Write => Release;
      Da: Data;
   end RA;

end Generic_Release_Acquire;
```

Example – Generic Release-Acquire Object (2/2)

```
package body Generic_Release_Acquire is

   concurrent body RA is

      procedure Write (D: Data) is
      begin
         Da := D;
         Ready := true;
      end Write:

      entry Get (D: out Data)
         until Ready is
         -- spin-lock until released, i.e., Ready = true;
         -- only sync. variables and constants allowed
         -- in guard expression
      begin
         D := Da;
      end Get;
   end RA;

end Generic_Release_Acquire;
```

API

```
package Memory_Model is

   type Memory_Order_Type is (
     Sequentially_Consistent,
     Relaxed,
     Acquire,
     Release);
   subtype Memory_Order_Success_Type is Memory_Order_Type;

   subtype Memory_Order_Failure_Type is Memory_Order_Type
     range Sequentially_Consistent .. Acquire;

   generic
      type Some_Synchronized_Type is private;
      with function Update return Some_Synchronized_Type;
      Read_Modify_Write_Variable: in out Some_Synchronized_Type
        with Read_Modify_Write;
      Memory_Order_Success: Memory_Order_Success_Type :=
        Sequentially_Consistent;
      Memory_Order_Failure: Memory_Order_Failure_Type :=
        Sequentially_Consistent;
   function Read_Modify_Write return Boolean;
end Memory_Model;
```

More examples. . .

. . . can be found in a Technical Report (cf. proceedings)

Conclusion and Future Work

- **concurrent objects** for encapsulating non-blocking data structures on a high abstraction level

Conclusion and Future Work

- **concurrent objects** for encapsulating non-blocking data structures on a high abstraction level
- **synchronized** and **read-modify-write** types which support the expression of memory ordering operations at a sufficient level of detail

Conclusion and Future Work

- **concurrent objects** for encapsulating non-blocking data structures on a high abstraction level
- **synchronized** and **read-modify-write** types which support the expression of memory ordering operations at a sufficient level of detail
- concurrent objects provide SC for programs without data races

Conclusion and Future Work

- **concurrent objects** for encapsulating non-blocking data structures on a high abstraction level
- **synchronized** and **read-modify-write** types which support the expression of memory ordering operations at a sufficient level of detail
- concurrent objects provide SC for programs without data races
- SC-for-DRF memory model is well-aligned with Ada's semantics for blocking synchronization via protected objects

Conclusion and Future Work

- **concurrent objects** for encapsulating non-blocking data structures on a high abstraction level
- **synchronized** and **read-modify-write** types which support the expression of memory ordering operations at a sufficient level of detail
- concurrent objects provide SC for programs without data races
- SC-for-DRF memory model is well-aligned with Ada's semantics for blocking synchronization via protected objects
- safe

Conclusion and Future Work

- **concurrent objects** for encapsulating non-blocking data structures on a high abstraction level
- **synchronized** and **read-modify-write** types which support the expression of memory ordering operations at a sufficient level of detail
- concurrent objects provide SC for programs without data races
- SC-for-DRF memory model is well-aligned with Ada's semantics for blocking synchronization via protected objects
- safe
- easy to understand

Conclusion and Future Work

- **concurrent objects** for encapsulating non-blocking data structures on a high abstraction level
- **synchronized** and **read-modify-write** types which support the expression of memory ordering operations at a sufficient level of detail
- concurrent objects provide SC for programs without data races
- SC-for-DRF memory model is well-aligned with Ada's semantics for blocking synchronization via protected objects
- safe
- easy to understand
- open issues: syntax

Conclusion and Future Work

- **concurrent objects** for encapsulating non-blocking data structures on a high abstraction level
- **synchronized** and **read-modify-write** types which support the expression of memory ordering operations at a sufficient level of detail
- concurrent objects provide SC for programs without data races
- SC-for-DRF memory model is well-aligned with Ada's semantics for blocking synchronization via protected objects
- safe
- easy to understand
- open issues: syntax, scheduling

Conclusion and Future Work

- **concurrent objects** for encapsulating non-blocking data structures on a high abstraction level
- **synchronized** and **read-modify-write** types which support the expression of memory ordering operations at a sufficient level of detail
- concurrent objects provide SC for programs without data races
- SC-for-DRF memory model is well-aligned with Ada's semantics for blocking synchronization via protected objects
- safe
- easy to understand
- open issues: syntax, scheduling, non-blocking barriers

Conclusion and Future Work

- **concurrent objects** for encapsulating non-blocking data structures on a high abstraction level
- **synchronized** and **read-modify-write** types which support the expression of memory ordering operations at a sufficient level of detail
- concurrent objects provide SC for programs without data races
- SC-for-DRF memory model is well-aligned with Ada's semantics for blocking synchronization via protected objects
- safe
- easy to understand
- open issues: syntax, scheduling, non-blocking barriers, integrating with other parallel programming features planned for Ada202x, ...